NASA GSFC
Data Systems Technology

Java™ Style Guide

July 1997

Sylvia B. Sheppard Date
Head, Software and Automations Systems Branch
Code 522

Julia D. Breed Date
Head, Applied Technology Development Section
Code 522.1

Goddard Space Flight Center
Greenbelt, Maryland

PREFACE

This document describes recommended practices and styles for programmers using the Java language in the Software and Automation Systems Branch, Code 522. Guidelines are based on generally recommended software engineering techniques, industry resources, and local convention.

This document is under the configuration management of the Software and Automation Systems Branch Configuration Control Board (CCB). Changes to this document shall be made by Documentation Change Notice (DCN), reflected in text by change bars, or by complete revision.

Requests for copies of this document, along with questions and proposed changes, should be addressed to:

Technology Support Office
Software and Automation Systems Branch, Code 522
Goddard Space Flight Center
Greenbelt, Maryland 20771

Or see our Web site at:
http://groucho.gsfc.nasa.gov/Code_520/Code_522/Documents/

This document assumes that the reader is familiar with the Java language and understands basic object-oriented concepts.

CHANGE INFORMATION PAGE

List of Effective Pages	
Page Number	**Issue**
Title	Original
Signature Page	Original
iii through x	Original
1-1 and 1-2	Original
2-1 and 2-11	Original
3-1 through 3-9	Original
4-1 through 4-3	Original
5-1	Original
I-2 and I-2	Original

Document History		
Issue	**Date**	**DCN No.**
Original	July 1997	N/A

ACKNOWLEDGMENTS

The following developers of the guidelines gave generously of their time and expertise during the process of development:

Troy Ames

Julia Breed

Carl Hostetter

Stephen Jonke

Jeremy Jones

Lisa Kane

Karl Mueller

Gregory Shirah

Mark Stirling

Robert Wiegand

TABLE OF CONTENTS

EXAMPLE CODE

SECTION 1
INTRODUCTION

1.1 PURPOSE

This document describes the style recommended by the Data Systems Technology Division for writing Java programs, where the goals are to produce code that is:

- Reliable
- Maintainable
  - Organized
  - Easy to understand
  - Well documented

1.2 AUDIENCE

This document was written specifically for programmers in the Data Systems Technology Division environment. This document assumes a working knowledge of Java, and focuses on describing good practices that will enhance the quality of the Java code.

1.3 DOCUMENT FORMAT

The following meanings are assigned to the formats included in this document:

Example code is included in shaded boxes.

Plain boxes are used to enclose standard formats and other related information.

Italics are used to differentiate the variable portion to be completed by the programmer from the standard format that should appear verbatim.

Italics are also used to visually separate brief (one-line) examples.

Bold text is used simply to highlight key words or phrases for the reader.

1.4 RELEVANT DOCUMENTS

[1] The Software and Automation Systems Branch C++ Style Guide, Version 2.0, DSTL-96-011
http://groucho.gsfc.nasa.gov/Code_520/Code_522/Documents/Cplus/

[2] The Java Programming Language, Arnold and Gosling
Addison-Wesley, 1996

[3] The Java Language Specification, Version 1.0
http://www.javasoft.com:81/docs/language_specification/

[4] The JavaBeans 1.0 API Specification
http://splash.javasoft.com/beans/spec.html

[5] Draft Java Coding Standard, Lea
http://g.oswego.edu/dl/html/javaCodingStd.html

[6] Java Optimization
http://www.cs.cmu.edu/~jch/java/optimization.html

SECTION 2
CONSISTENT FORMATTING

The guidelines in this section are written to improve the consistency of formatting of Java programming style within the Branch. This will ease the job of maintenance, and will also make it easier for a programmer to transfer from one project to another.

2.1 WHITE SPACE

Adding **white space** in the form of blank lines, spaces, and indentation significantly improves the readability of code.

2.1.1 BLANK LINES

There should be at least one blank line between methods. Within a method careful use of **blank lines** between code "paragraphs" can greatly enhance readability by making the logical structure of a sequence of lines more obvious. Using blank lines to create paragraphs in code or comments can make programs more understandable. The following example illustrates how the use of blank lines helps break up lines of text into meaningful portions.

Example 2.1.1a - Code Paragraphing

```
public void joinGroupSucceeded(JoinGroupSuccessEvent e)
{
    Group g = getPendingGroup(e.getGroupName());

    if (g != null)
    {
        // Remove g from the pending list
        fPendingGroupTable.remove(g.getName());

        // Add the group to the joined group list
        fJoinedGroupTable.put(g.getName(), g);
    }

    // Notify listeners of the event:

    Vector listeners = null;
    synchronized(this)
    {
        listeners = (Vector) fJoinGroupSuccessListeners.clone();
    }

    for (int i = 0; i < listeners.size(); ++i)
    {
        ((JoinGroupSuccessListener) listeners.elementAt(i)).joinGroupSucceeded(e);
    }
}
```

However, overuse of blank lines can defeat the purpose of grouping and can actually reduce readability. Therefore, a single blank line should be used to separate sections of code within a method.

2.1.2 SPACING

Appropriate **spacing** enhances the readability of lexical elements.

a. Do not put space around the **primary operators**: . and []:
```
obj.m     a[i]
```

b. Do not put a space before **parentheses** following method names.
```
exp(2, x)
```

c. Do not put spaces between **unary operators** and their operands:
```
!p     -b     ++i     -n
```

d. **Casts** are the only exception. *Do put a space* between a cast and its operand:
```
(Cloneable) object
```

e. Always put spaces around **assignment operators:**
```
c1 = c2
```

f. Always put space around **conditional operators:**
```
z = (a > b) ? a : b;
```

g. **Commas** should have one space (or a new line) after them:
```
stream.read(buffer, 0, 255)
```

h. **Semicolons** should have one space (or a new line) after them:
```
for (i = 0; i < n; ++i)
```

i. For **other operators**, generally put one space on either side of the operator:
```
x + y     a < b && b < c
```

2.1.3 INDENTATION

Indentation should be used to show the logical structure of code. Research has shown that **four spaces** are the optimum indent for readability and maintainability. Tabs should be used for indentation, and it is recommended that they be set to four spaces for displays. Align groups of variable declarations (using tabs) so that the first letter of each variable name is in the same column.

2.1.4 CONTINUATION LINES

Line length should not exceed 79 characters. Statements that continue over more than one line should be indented each line after the first line with **two additional tabs**. This will differentiate the continuation portion of the statement from the encapsulated body of a block.

a. Strings of conditional operators that will not fit on one line should be divided into separate lines, breaking before the logical operators in the following format:

Example 2.1.4a - Strings of Conditional Operators

```
if (listenerClass.isInstance(listener)
        && arguments.length == 1
        && arguments[0].getName().equals(eventClassName))
{
    // Wrap event object in array
    Object[] invokeArgs = new Object[1];
    invokeArgs[0] = this;

    // Invoke the method
    method.invoke(listener, invokeArgs);
}
```

b. Method calls and declarations that continue over more than one line should be presented one argument per line, followed by a comma in the following format. One benefit of this is that an in-line comment can be added after each argument to describe its purpose. If argument names are short, and their meanings are simple, they can be combined on one line at the programmer's discretion.

Example 2.1.4b - Method Call

```
JoinGroupSuccessEvent successEvent = new JoinGroupSuccessEvent(
        this,                   // source workplace instance
        getName(),              // source name
        member.getName(),       // destination member name
        group.getName());       // destination group name
```

c. A method definition's return type should go on the same line as the method name.

2.1.5 BRACES AND PARENTHESES

Compound statements, also known as blocks, are lists of statements enclosed in braces. The recommended brace style is the **Braces-Stand-Alone** method. Braces should be placed on separate lines and aligned with their contents indented one tab which is four spaces. This style allows for easier pairing of the braces.

Example 2.1.5a - Braces-Stand-Alone Method

```
int total = 0;
for (int i = 0; i < elements; i++)
{
    total += i;
    sum[i] = total;
}
```

a. Although Java does not require braces around single statements, braces should be used for single statement blocks to help improve the readability and maintainability of the code. If braces are not used in single statement blocks the risk of maintenance errors is increased. Maintenance programmers may add a statement within the block, but may forget to add braces.

Example 2.1.5b - Braces Improve Readability

```
for (int i = 0; i < editMenu.getItemCount(); ++i)
{
    if (editMenu.getItem(i).getLabel().equals("Settings...")
            && fUser.isAdministrator())
    {
        editMenu.remove(i);
    }
}
```

Example 2.1.5c - No Braces - Difficult to Read

```
for (int i = 0; i < editMenu.getItemCount(); ++i)
    if (editMenu.getItem(i).getLabel().equals("Settings...")
            && fUser.isAdministrator())
        editMenu.remove(i);
```

b. Because the else part of an if-else statement is optional, omitting the 'else' from a nested if sequence can result in ambiguity. Therefore, always use braces to avoid confusion and to make certain that the code compiles the way it was intended. In the following example, (2.1.5d) the same code is shown both with and without braces. The first example will produce the results desired. The second example will not produce the results desired because the 'else' will be paired with the second 'if' instead of the first.

Example 2.1.5d - Use of Braces

recommended:

```
if (fConnected)
{
    for (int i = 0; i < n; i++)
    {
        if (s[i] > 0)
        {
            fCounts.add(s[i]);
        }
    }
}
else    // CORRECT -- braces force proper association
{
    System.out.println("Error: not connected.");
}
```

not recommended:

```
if (fConnected)
    for (int i = 0; i < n; i++)
        if (s[i] > 0)
        {
            fCounts.add(s[i]);
        }
else    // WRONG -- the compiler will match to closest else-less if
    System.out.println("Error: not connected.");
```

c. If a loop statement has a dummy body, opening and closing brackets should be added on separate lines. It is good practice to add a comment stating that the dummy body is deliberate.

Example 2.1.5e - Dummy Body

```
    while (fTable.loadElement())
    {
        // dummy body
    }
```

d. Insert a space between reserved words and their opening parentheses.

2.2 COMMENTS

Comments in the code can provide information that a person could not discern simply by reading the code. In addition to the standard required comment blocks (see Section 4), comments can be added at

many different levels. Comments can be written in several styles depending on their purpose and length. Use comments to **add information** for the reader or to **highlight sections** of code. Comments should not paraphrase the code. This section covers two types of comments: documentation comments and code comments.

2.2.1 DOCUMENTATION COMMENTS

Java programs can include special documentation comments in their source code. The **javadoc** tool extracts these comments from the code and produces Web pages that document the code. This is the preferred way to create documentation for Java code. Documentation comments should be used to comment classes, interfaces, methods, and fields.

Example 2.2a - Documentation Comment

```
/**
 * Description of the object being commented.  The description may
 * span multiple lines.
 *
 * @tags appropriate to the type of block are placed here, one per line.
 * (see the Java Language Specification, section 18.4 for a list of tags)
**/
```

General guidelines for using documentation comments are as follows:

a. A documentation comment should precede the code that it refers to.

b. Use the HTML tags "<PRE>" and "</PRE>" when including sample code in a documentation comment. The tags ensure that the code is displayed correctly in the **javadoc** generated files. [3]

c. Other HTML tags, such as "<B>" and "</B>" for boldface, may be used. However, the heading tags ("<H1>", "<H2>", etc.) and horizontal line tag ("<HR>") are reserved for use by **javadoc** and should not be used.

d. The first sentence of each documentation comment should be a summary sentence, containing a concise but complete description of the declared entity. This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first tagline. [3]. The **javadoc** parser will include only the summary sentence in the summary section of the document, but will include the entire comment in the details section.

e. Remember that white space formatting is ignored by the parser. Even if you have paragraphs separated by blank lines in your comment, it will appear as one big paragraph in the HTML output. Use the HTML tags "<P>" and "
" to separate sections of your comment.

2.2.2 CODE COMMENTS

2.2.2.1 Code Comment Formats

There are several different formats of code comments:

a. **Boxed Comments** - Used for standard format comment blocks (see Section 4).

Example 2.2b - Boxed Comment

```
//===========================================================================
// This code was developed by NASA, Goddard Space Flight Center, Code
// XXX for the Project Name Project
//===========================================================================
```

b. **Section Separators** - Used to visually separate portions of a program or comment block.

Example 2.2c - Section Separator

```
//---------------------------------------------------------------------------
```

c. **Block comments** - Used at the beginning of a section of code as a narrative description of that portion of the code.

Example 2.2d - Block Comment

```
// This is a block comment. The comment should be written in full
// sentences with correct punctuation, etc. Use this form of comment
// when more than one sentence is required
```

d. **In-Line Comments** - Written on the same line as the code or data definition they describe. These comments should be tabbed over far enough to separate them from the code statements. In addition, if more than one short comment appears in a block of code or data definition, they should all be started from the same tab position.

Example 2.2e - In-Line Comments

```
Matrix33    transformationMatrix;  // matrix converting from GCI to BCS
int         numModels;             // number of attitude models used
String      name;                  // name of current attitude model
```

2.2.2.2 General Guidelines

General guidelines for using code comments are as follows:

a. Declare each internal (local) variable on a separate line followed by an in-line comment if necessary. Loop indices and other such insignificant variables are an exception to this rule. They can all be listed on the same line with one comment.

b. Comments should describe blocks of code rather than individual lines of code (see Example 2.2d).

c. Comments should be written at the same level of indentation as the code they describe (see Example 2.2g).

Example 2.2f - Block Comment vs. In-line Comment

preferred style:

```
    //
    // Main sequence: get and process all user requests
    //

    while (!isFinished())
    {
        inquire();
        process();
    }
```

not recommended:

```
    while (!isFinished())         // Main sequence:
    {                             //
        inquire();                // Get user request
        process();                // And carry it out
    }                             // As long as possible
```

Example 2.2g - Comment Indentation

```
    //
    // Main sequence: get and process all user requests
    //

    while (!isFinished())
    {
        inquire();

        if (requestCode != 0)
        {
            // If the request code is non-zero, then perform
            // intermediate processing to generate request information
            generateRequestInfo(requestCode);
        }

        process();
    }
```

2.3 STANDARD NAMING CONVENTIONS

2.3.1 NAME FORMATS

Classes and Interfaces: Capitalize the first letter of each word.
Example: CustomerSchedule

Variables: Capitalize the first letter of each word except the first word.
Example: numberOfStars

Member Fields: Begin with the letter "**f**", then conform to the variable format.
Example: fTimeOfDay

Methods: Capitalize the first letter of each word except the first word.
Example: getValue

Static Members of a Class: Begin with the letter "**s**", then conform to variable format.
Example: sTimeOfDay

Constants: All uppercase, with words separated by underscore "_" characters.
Example: MIN_VALUE

Names Containing Acronyms: Follow the above conventions.
Example: NasaIdentifier (class name)

Names Containing Proper Nouns: Follow the above conventions.
Example: kennedyCode (variable name)

2.3.2 NAME CONVENTIONS

To improve consistency with the Java API and to provide support for JavaBeans, names and verb phrases should also obey the following conventions:

a. Methods to get and set an attribute should be named **getX** and **setX**, where X is the attribute.
Examples: getFont and setFont of class java.awt.Component

b. A method that tests a boolean condition should be named **isX** where X is the condition to test.
Example: isEnabled of class java.awt.Component

c. A method that converts its object to a new format should be named **toX** where X is the new format.
Example: toString of class java.lang.Object

d. Subclasses of java.util.EventObject should be named **XEvent**, where X is the event name.
Example: ButtonPressEvent

e. Subclasses of java.util.EventListener should be named **XListener**, where X is the event type that the listener is interested in, minus the "Event" suffix.
Example: ButtonPressListener, where ButtonPressEvent is the event of interest

f. Event sources that allow registration of event listeners should name their registration methods **addX** and **removeX**, where X is the event listener type.

Examples: addButtonPressListener, removeButtonPressListener

2.3.3 SHORT NAMES

Some standard **short names** for code elements are listed in the table below. While use of these names is acceptable if their meaning is clear, more explicit names are recommended.

Standard Short Names:

c	characters
i, j, k	indices
m, n	counters
o	objects
e	exceptions
s	strings

2.3.4 GENERAL GUIDELINES FOR VARIABLE NAMES

The following guidelines should be adhered to before writing code:

a. Names should be nouns or noun phrases.

b. Use meaningful names. Longer names improve readability and clarity.

c. Avoid the use of underscores, except in constants.

d. Avoid abbreviations. However, if required, follow a uniform scheme.

e. Names should differ by at least two characters. For example, "systst" and "sysstst" are easily confused.

f. Do not rely on letter case to make a name unique.

g. Avoid using gratuitous modifiers (e.g., the and my) as the first word of a name.

h. In separate functions, do not use identical variable names for variables that do not have identical meaning. Using the same variable name if the meaning of two variables are only similar or coincidental can cause confusion to the reader.

2.3.5 PACKAGE NAMING CONVENTIONS

The Java Language Specification (see Section 1.4) suggests a standard convention for naming packages. It is strongly recommended that package names adhere to this standard to avoid naming conflicts.

a. The first components of a package name should be the components of the organization's domain name in reverse order, with the first component in all uppercase. (e.g., GOV.nasa.gsfc)

b. If classes belong to a specific project then append a component name corresponding to the name of the project. (e.g., GOV.nasa.gsfc.workplace)

c. If classes are intended to be used across multiple projects, then append a “reuse” component to their package name. (e.g., GOV.nasa.gsfc.reuse)

2.4 LITERALS

Literals should adhere to the following guidelines:

a. Floating point numbers should have at least one number on each side of the decimal point:

```
0.5                    5.0                    1.0e+36
```

b. Hexadecimal numbers should use 0x (zero, lower-case x) and upper case A-F:

```
0x123                  0xFFF
```

SECTION 3
FILE ORGANIZATION

This section discusses the organization of Java source code. It describes how to package related source files together and how to organize code within individual source files.

3.1 PACKAGES

A Java program consists of one or more Java classes that are normally stored in source files. Packages organize the source files into groups of related files. All source files, and thus all classes, belong to a package. For simple programs a package may not be specified. In this case, classes become members of the unnamed package.

Files within a package should be organized using a directory structure where package components are represented as subdirectories. Organize files by class with one class definition per file. The file should have the exact name as the class/interface, plus a ".java" extension.

3.2 README FILE

A **README.html** file should be used to explain what the program does and how it is organized and to document issues for the program as a whole. For example, a README.html file might include the following:

a. How to build and run the software (i.e., version of Java needed, additional required packages).

b. How to configure the environment.

c. Pointers to additional documentation. If providing source, include a link to the Javadoc.

3.3 FILE PROLOG

This prolog should appear at the beginning of all source files. The format should also be used for other files related to the program, such as scripts and Makefiles, although the comment indicator ('//') must be changed.

Example 3.3a - File Prolog

```
//=== File Prolog ============================================================
//    This code was developed by NASA, Goddard Space Flight Center, Code XXX
//    for the Project Name project.
//
//--- Notes -------------------------------------------------------------------
//    Anything relevant about the items in this file, including document
//    references, assumptions, constraints, restrictions, abnormal termination
//    conditions, etc.
//
//--- Development History -----------------------------------------------------
//    Date         Author       Reference
//    Description
//
//    Date         Author       Reference
//    Description
//
//       Etc.
//
```

```
//--- Warning ----------------------------------------------------------------
//    This software is property of the National Aeronautics and Space
//    Administration.  Unauthorized use or duplication of this software is
//    strictly prohibited.  Authorized users are subject to the following
//    restrictions:
//    *   Neither the author, their corporation, nor NASA is responsible for
//        any consequence of the use of this software.
//    *   The origin of this software must not be misrepresented either by
//        explicit claim or by omission.
//    *   Altered versions of this software must be plainly marked as such.
//    *   This notice may not be removed or altered.
//
//=== End File Prolog ============================================================
```

3.4 ORDER OF CONTENTS

The items contained within a source file should be presented in the following order:

```
File prolog (see 3.3)
Package declaration
Import statements (core packages)
Import statements (user-defined packages)
Class/interface prolog
Class/interface definition
```

3.5 CLASSES AND INTERFACES

Each class or interface shall be preceded by a documentation comment of the following format. The @deprecated and @see lines are to be used only if applicable.

Example 3.5a - Classes and Interfaces

```
/**
 * Description of the class.  Include usage instructions.  Embed code fragments
 * in <PRE></PRE> tags.
 *
 * <P>This code was developed by NASA, Goddard Space Flight Center, Code XXX
 * for the Project Name project.
 *
 * @version        date of submission
 * @author         author's name
 *
 * @deprecated     (add this if the class is superceded by a new class)
 * @see            name of a related class or interface, fully qualified
 * @see            relatedClassName#methodInClass
 * @see            URL (for document references)
**/
```

3.6 METHODS

Keep methods simple. Divide complex tasks into multiple methods. Precede each method with a documentation comment of the following format. The @deprecated and @see lines are to be used only if applicable.

Example 3.6a - Methods

```
/**
 * Describe the method:  what it does, its purpose, effects, usage instructions, and
 * implementation notes.
 *
 * @param           argumentName        description of argument
 * @param           argumentName        description of argument
   (etc.)
 *
 * @return          description of return value (omit if returns void)
 *
 * @exception       exceptionName       description of exception that method throws
 * @exception       exceptionName       description of exception that method throws
   (etc.)
 *
 * @deprecated              (add this if the method is superceded by a new method)
 * @see                     #otherMethodInThisClass
 * @see                     relatedClassName#methodInClass
 * @see                     URL (for document references)
**/
```

3.7 TAGGED PARAGRAPHS

@version Tag	May be used in documentation comments for class and interface declarations i.e., @version 493.9.1beta
@author Tag	May be used in documentation comments for class and interface declarations i.e., @author Jeremy Jones
@param Tag	May be used in documentation comments for method and constructor declarations i.e., @param file the file to be searched @param pattern the pattern to be matched during the search @param count the number of lines to print for each match
@return Tag	May be used in documentation comments for declarations of methods whose result type is not void i.e., @return the number of widgets that pass the quality test
@exception Tag	May be used in documentation comments for method and constructor declarations i.e., @exception IndexOutOfBoundsException the matrix is too large @exception UnflangedWidgetException the widget does not have a flange, or its flange has size zero @exception java.io.FileNotFoundException the file does not exist
@deprecated Tag	Class/method is superceded by another class/method.
@see Tag	May be used in any documentation comment to indicate a cross-reference to a class, interface, method, constructor, field, or URL i.e.,. @see java.lang.String @see String @see java.io.InputStream; @see String#equals @see java.lang.Object#wait(int) @see java.io.RandomAccessFile#RandomAccessFile(File, String) @see Character#MAX_RADIX @see Java Spec

3.8 SAMPLE SOURCE FILE

Example 3.8a - Sample Source File

```
//=== File Prolog ============================================================
//  This code was developed for NASA, Goddard Space Flight Center, Code 520
//  for the Instrument Remote Control (IRC) project.
//
//--- Notes -------------------------------------------------------------------
//  This class requires JDK version 1.1 or later.
//
//--- Development History -----------------------------------------------------
//
//  11/01/96    K. Campbell/522
//
//      Initial version.
//
//  02/01/97    J. Jones/522
//
//      Converted class to comply with JavaBeans conventions.
//      Now uses serialization to send/receive event objects.
//
//--- Warning -----------------------------------------------------------------
//  This software is property of the National Aeronautics and Space
//  Administration.  Unauthorized use or duplication of this software is
//  strictly prohibited.  Authorized users are subject to the following
//  restrictions:
//  *   Neither the author, their corporation, nor NASA is responsible for
//      any consequence of the use of this software.
//  *   The origin of this software must not be misrepresented either by
//      explicit claim or by omission.
//  *   Altered versions of this software must be plainly marked as such.
//  *   This notice may not be removed or altered.
//
//=== End File Prolog ========================================================

package GOV.nasa.gsfc.workplace;

import java.net.Socket;
import java.net.UnknownHostException;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;

/**
 * This class represents a connection between two Workplace members.
 * It is used to send event objects to a remote member.  It is also used
 * to receive an event object from a remote member.  When that occurs,
 * TCPConnection will instruct the event object to call the appropriate
 * Listener method on the TCPConnection's parent object.
 *
 * <P>This code was developed for NASA, Goddard Space Flight Center, Code 520
 * for the Instrument Remote Control (IRC) project.
 *
 * @version             02/01/97
 * @author              J. Jones
 *
 * @see                 GOV.nasa.gsfc.workplace.Connection
**/
public class TCPConnection extends Thread implements Connection
{
    private static final String BEGIN_SYNC = "_WP_EVENT_BEGIN_";
    private static final String END_SYNC = "_WP_EVENT_END_";

    private Socket              fSocket;    // the actual socket connection
    private WorkplaceListener   fParent;    // notify when data is received
    private ObjectInputStream   fIn;        // input object stream
```

```
    private ObjectOutputStream  fOut;       // output object stream


    /**
     * Creates a TCPConnection from an existing Socket.  The socket already
     * has an open connection, so TCPConnection immediately starts
     * its thread and waits for data.
     *
     * @param   s           Existing socket connection
     * @param   parent      Notify this object when data is received
     *
    **/
    public TCPConnection(Socket s, WorkplaceListener parent)
    {
        fSocket = s;
        fParent = parent;
        getIOStreams();
        start();
    }



    /**
     * Creates a TCPConnection given the address of the remote member.
     * It then attempts to connect to the remote member.
     *
     * @param   address     (host, port) address of the remote member
     * @param   parent      Notify this object when data is received
     *
    **/
    public TCPConnection(TCPAddress address, WorkplaceListener parent)
    {
        fParent = parent;

        try
        {
            open(address);
        }
        catch (IOException e)
        {
        }
    }



    /**
     * Creates an empty TCPConnection.  open(address) must be called
     * later to initialize a connection to a remote member.
     *
     * @param   parent      Notify this object when data is received
     *
    **/
    public TCPConnection(WorkplaceListener parent)
    {
        fParent = parent;
    }



    /**
     * Attempts to establish a connection to a remote host and port.
     * If successful, starts the TCPConnection thread and waits for data.
     *
     * @param   address     (host, port) address of the remote member
     *
     * @exception   IOException     an error occurred in opening the connection
    **/
    public void open(Address address) throws IOException
    {
        System.out.println("Opening connection to "
                + ((TCPAddress)address).getHost()
```

```
                + ":" + ((TCPAddress)address).getPort());
        try
        {
             fSocket = new Socket(((TCPAddress)address).getHost(),
                  ((TCPAddress)address).getPort());
        }
        catch (UnknownHostException e)
        {
            System.out.println("Unknown host.");
            fSocket = null;
        }
        catch (IOException e)
        {
            fSocket = null;
        }

       if (fSocket == null)
        {
            System.out.println("Error in opening connection.");
            throw (new IOException());
        }

        System.out.println("Apparently connected.");

        getIOStreams();

        start();
    }



    /**
     * Closes the connection to the remote object and stops the thread.
    **/
    public void close()
    {
        stop();

        try
        {
            fSocket.close();
        }
        catch (IOException e)
        {
        }
    }



    /**
     * Attempts to get I/O streams for the existing socket connection.
    **/
    private void getIOStreams()
    {
        try
        {
            fIn = new ObjectInputStream(new BufferedInputStream
                    (fSocket.getInputStream()));
            fOut = new ObjectOutputStream(new BufferedOutputStream
                    (fSocket.getOutputStream()));
        }
        catch (IOException e)
        {
            System.out.println("Error in getting streams.");
        }
    }



    /**
      * This routine waits for incoming data from the socket connection.
```

```
     * It reads event objects from the input stream and notifies the
      * parent listener.
     **/
     public void run()
     {
         while (true)
         {
             try
             {
                 // Read the begin sync
                 String beginSync = fIn.readUTF();

                 // Don't try to read object if incorrect begin sync
                 if (beginSync.equals(BEGIN_SYNC))
                 {
                     try
                     {
                         System.out.println("Receiving incoming event...");

                         WorkplaceEvent event = (WorkplaceEvent) fIn.readObject();

                         // Notify the parent listener
                         event.notifyListener(fParent);
                     }
                    catch (ClassNotFoundException e)
                     {
                         System.out.println("Unable to create instance of event");
                     }
                 }

                 // Read the end sync
                 String endSync = fIn.readUTF();

                 // Stop if incorrect end sync
                 if (!endSync.equals(END_SYNC))
                 {
                     throw (new IOException("Unable to read Workplace end sync
token"));
                 }
             }
             catch (IOException e)
             {
                 System.out.println("IO Exception occurred in reading incoming
event");
                 stop();
             }
         }
     }



     /**
      * Writes an event object to the output stream.
      *
      * @param   event   object to write
      *
     **/
     public synchronized void send(WorkplaceEvent event)
     {
         try
         {
             System.out.println("Sending an event: " + event.toString());

             // Write begin sync string
             fOut.writeUTF(BEGIN_SYNC);

             // Write the serialized object to the stream
             fOut.writeObject(event);

             // Write end sync string
             fOut.writeUTF(END_SYNC);
```

```
            fOut.flush();
        }
        catch (IOException e)
        {
            System.out.println("Write failed.");
        }
    }



    /**
     * Returns the current state of the connection as a ConnectionState
     * object.  Note:  the current implementation of this method simply
     * returns a default ConnectionState object.
     *
     * @return      object representing the state of the current connection
     *
    **/
    public ConnectionState getState()
    {
        return new ConnectionState();
    }
}
```

SECTION 4
USE OF LANGUAGE CONSTRUCTS

This section contains rules that programmers should follow when using various Java language features.

4.1 IMPORT

a. Only import classes and interfaces that are actually being used rather than using * forms of import.

b. Group related imports together with core packages first followed by user-defined packages.

4.2 METHODS

a. A **single return statement** at the end of a function creates a single, known point that is passed through at the termination of function execution. **Multiple returns** in a single unit should be avoided as much as possible, unless the use of a single return would cause the code to be difficult to understand or maintain.

b. In the case of a **thrown exception**, the method should attempt to leave a consistent state before throwing the exception. (This includes making objects available for garbage collection and setting appropriate state variables, as necessary.)

4.3 VARIABLES

a. **Local variables** should be declared at the level at which they are needed. For example, if a variable is used throughout the procedure, it should be declared at the beginning of the procedure. If a variable is used only in a computational block, it may be declared at the top of that block. Local variables should be **initialized** when they are declared.

b. Do not use internal variable declarations that override declarations at higher levels; these are known as **hidden variables**.

c. Place fields at the **beginning** of the class, in order of accessibility (e.g., public, protected, package, private).

d. Never declare fields as public.

e. Always explicitly **initialize** variables when declaring them. Do not rely on implicit initializers.

f. Minimize the use of static fields, except for static final constants.

g. Prefer the `Type[] arrayName` form of array declaration instead of the `Type arrayName[]` form.

h. Assign `null` to any object references that are no longer being used. This enables garbage collection.

4.4 LITERALS

a. Use named constants instead of embedded literals, except in trivial cases.

4.5 TYPE CONVERSIONS AND CASTS

a. Type conversions occur by default when different primitive types are mixed in an arithmetic expression across an assignment operator. Use the cast operator to make type conversions **explicit** rather than implicit.

b. Avoid casting an object to its base type. It is unnecessary and can create confusion.

4.6 OPERATORS AND EXPRESSIONS

a. Use **side-effects** within expressions sparingly. No more than one operator with a side-effect (=, ++, --) should appear within an expression. It is easy to misunderstand the rules for compilation and get side-effects compiled in the wrong order e.g.,

```
if ((a < b) && (c == d++))
```

d will “only” be incremented if a is less than b.

Avoid using side-effect operators within relational expressions. Even if the operators do what the author intended, subsequent reusers may question what the desired side-effect was.

b. Avoid the use of embedded assignments. Example 4.7a illustrates the relative readability of embedded assignments.

Example 4.7a - Embedded Assignment

not recommended:

```
if ((total = getTotal()) == 10)
{
    System.out.println("goal achieved");
}
```

recommended:

```
total = getTotal();
if (total == 10)
{
    System.out.println("goal achieved");
}
```

c. In Java, conditional expressions allow evaluation of expressions and assignment of results in a **shorthand way**. While some conditional expressions seem very natural, others do not, and their use should be avoided. The following expression, for example, is not as readable as the one above and would not be as easy to maintain. It should be broken into individual statements.

Example 4.8a - Complex Conditional Expression

not recommended

```
c = (a == b) ? d + f(a) : f(b) - d;
```

4.7 CONTROL FLOW STATEMENTS

a. For readability, use the following format for switch statements:

```
switch (expression)
{
    case aaa:
        statement[s]
        break;

    case bbb:    // fall through
    case ccc:
        statement[s]
        break;

    default:
        statement[s]
        break;
}
```

b. Note that the **fall-through feature** of the Java switch statement should be commented for future maintenance.

c. All switch statements should have a **default case**. The default case should be last and does not require a break, but it is a good idea to put one there anyway for consistency.

d. Avoid the use of labels.

e. Avoid the use of break and continue except when using breaks in switch statements.

4.8 THREADS

a. If a method invokes `wait`, document that fact in the method prolog.

b. When a `wait` finishes, it does not know if the condition it was waiting for is true or not, therefore always embed `wait` statements in while loops that re-wait if the condition is false.

SECTION 5
TIPS AND TECHNIQUES

This section contains various suggestions that can be used to improve the quality of Java code.

5.1 CLASSES

a. Consider whether a class should implement the `Cloneable` and/or `Serializable` interfaces.

b. If a class implements `Cloneable`, ensure that its implementation of clone() performs a deep copy.

c. Consider defining a default (no-argument) constructor so that instances can be created via Class.newInstance() and Beans.instantiate().

d. Use interfaces to separate functionality from implementation.

   1. Code written in terms of the abstract (interface) type does not need to change when the implementation class changes.

   2. Different implementations of the interface can be used, even at the same time.

   3. Other (unforeseen) classes can implement the interface if it becomes necessary, while they are only allowed to inherit from a single class.

   4. The interface provides a distinct location where the behavior is defined.

e. A class that overrides Object.equals() should also override Object.hashCode(), and vice-versa. This allows objects of the class to be properly inserted into container objects.

5.2 THREADS

a. Do not depend upon a particular implementation of threads since the virtual machine specification does not specify how threads will be scheduled.

b. In order to promote reuse, always assume that a method may be called by multiple threads. Synchronization should be used on all methods that change or read the internal state of the object.

5.3 PORTABILITY

a. Consider detailed optimizations only on computers where they prove necessary. Optimized code is often obscure. Optimizations for one computer may produce worse code on another. Document code that is obscure due to performance optimizations and isolate the optimizations as much as possible.

b. Native methods are inherently nonportable. Organize source files so that the computer-independent code and the computer-dependent code are in separate classes. If the program is moved to a new computer, it will be clear which classes need to be changed for the new platform.

5.4 PERFORMANCE

a. When performance is important, as in real-time systems, use techniques to enhance performance. If the code becomes "tricky" (i.e., possibly unclear), add comments to aid the reader.

b. Use buffered I/O streams to improve the performance of stream reads and writes.

c. Enable compiler optimizations when compiling performance-critical classes. This will enable inlining of static, final and private methods.